加快排序文档的剪枝决策树和分块的方法 *

李卫疆，常　伟，余正涛

(昆明理工大学 信息工程与自动化学院，昆明 650500)

摘　要：近年来，越来越多的人从互联网上检索自己需要的信息。检索系统利用排名学习算法从训练集中产生一个排名模型。而检索数据需要的时间则是检索系统的一种重要研究方向。为了减少检索的时间，对排名模型的剪枝策略和缓存进行了研究。利用决策树的冗余特性和高速缓冲存储器，提出了剪枝决策树模型和分块算法。最后，在两个公开的数据集上进行了实验，主要关注了是否可以在不影响模型效果下，提高排名模型的效率的问题。实验结果表明剪枝决策树模型和分块算法可以有效地减少每个查询的排名时间。

关键词：排名学习；缓存；效率；剪枝

中图分类号：TP311　　doi: 10.19734/j.issn.1001-3695.2018.05.0443

Method of pruning decision tree model and block-wise to speed-up ranking candidate documents

Li Weijiang, Chang Wei, Yu Zhengtao

(*School of Information Engineering & Automation, Kunming University of Science & Technology, Kunming* 650500, *China*)

Abstract: Recently, more and more people start from the Internet to retrieve the information we need. The retrieval system uses learning to rank (LtR) algorithms to produce a ranking model from public available training sets. The time required to retrieve data is an important research direction of the retrieval system. In order to reduce the retrieval time, the pruning strategy of ranking model and cache were studied. Using the redundancy characteristics of the decision tree and the cache, a pruning decision tree model and block-wise algorithm are proposed. Finally, this paper aims to answer the question that whether it can improve the efficiency of the ranking model without affecting the effectiveness of model. Experiments on two publicly dataset show that the pruning decision tree model and block-wise algorithm can effectively reduce the scoring time per query.

Key words: learning to rank; cache; efficiency; pruning

0　引言

在信息检索中，根据相关性准则，对用户查询的相关内容进行排序是一个基础且重要的问题。一个名叫排名学习[1,2]的研究领域显示利用机器学习方法可以有效解决排名问题。一种机器学习算法用于从一个包含相关性级别的查询文档对的数据集中训练出一个排名模型。将基于决策树的集合体的排名学习模型用于排序 Web 搜索引擎返回的查询结果是非常有效的[3,4]。

一个复杂的排名体系结构通常由检索候选文档和重新排序候选文档两部分组成[5]。第一个阶段是从一个倒排索引中检索出一个足够大的候选文档集（CDq），这个候选文档集是可能与用户的查询相关，也有可能不相关。|RDq|表示在最终页面上显示的相关文档的个数，并且|CDq|>>|RDq|。这个阶段的目的是提高召回率。这个检索出候选文档集的初步的过滤器通常是一种简单而快速的基本算法，如 BM25 算法[6]、布尔模型和扩展的布尔模型。这个排名学习模型被用于第二个阶段，排名学习模型用于重新排序这个候选文档集。这个阶段的目的是提高准确度。最终，在排序候选文档集之后，前|RDq|文档显示在最终页面上。在这两阶段体系架构中，因为大量的查询和用户对响应时间的要求，所以用于再次排序候选文档集的时间是非常有限的。

现代搜索引擎对于响应用户查询有非常严格的时间限制。探索使用新的策略和技术去减少排名查询结果的时间是一个紧急的研究话题。因此，一些研究者已经开始去探索减少排名时间的优化技术。例如 Lucchese 等人[7]提出一种名叫 QuickScorer 的排名算法。这种算法采用比特向量的方式去展现一个决策树

收稿日期：2018-05-06；**修回日期：**2018-07-18　　**基金项目：**国家自然科学基金资助项目（61363045）；云南省自然科学基金重点资助项目（2013FA130）；科技部中青年科技创新领军人才资助项目（2014HE001）

作者简介：李卫疆（1969-），男，副教授，博士，主要研究方向为信息检索、自然语言处理（hrbrichard@126.com）；常伟（1993-），男，硕士研究生，主要研究方向为信息检索；余正涛（1970-），男，教授，博士，主要研究方向为自然语言处理、信息检索.

集合体和新的访问模式。Asadi 等人[8]通过将控制依赖转换为数据依赖的方式来遍历这个决策树集合体。Lucchese 等人[9]提出一种基于排名特征的框架，用于去扩展原始特征集，以减少排名文档的时间。Lucchese 等人[10]提出了一种 V-QuickScorer 的算法，这种算法利用现代 CPU 的 SIMD 指令集去向量化处理多个文档排名。周栋等人[11]提出了个性化跨语言信息检索中结果重排序研究。

为了减少每个查询的排名时间，本文提出了剪枝决策树模型和分块算法的方法。该方法整合了剪枝和缓存两种技术。一方面，当计算一篇文档的得分时候，要去遍历集合体中所有的决策树。所以排名文档的时间与决策树的个数成正比。本文使用剪枝技术在不影响模型的效果情况下去减少决策树的个数。另一方面，分块技术将会充分利用缓存时间局部性原理[12]，以便减少排名查询的时间。本文的实验显示在效率方面该方法要比最先进基线排名算法表现的更好，如 StructPlus、VPRED。本文主要贡献如下：

a）每一篇文档需要遍历集合体中所有的决策树，每一个文档将有一个用于排名文档的得分。预测一篇文档得分的时间与树的个数成正比关系。所以在不影响排名模型的效果下，使用剪枝技术去减少决策树个数，就可以减少排名文档的时间。

b）因为内存访问延迟比缓存访问延迟慢几个数量级。为了更快的排序文档，本文使用缓存技术去优化文档的遍历。一个候选文档集和一个决策树集合体的大小可能会超过缓存的容量，就会导致缓存的内容被频繁的替换。为了充分利用缓存的时间局部性，本文使用分块技术分别将一个候选文档集和一个决策树集合体划分到各自几个块中，就可以进一步减少排名时间。

1　相关工作和背景

越来越多的文档出现在因特网中。对于用户的查询，初步的过滤器算法将会从互联网上检索出更多的相关文档，这就导致排名候选文档需要更多的时间。

如今，排名学习算法已经被普遍用于解决大多数排名问题。这有两种最有效果的排名学习算法。分别是 GBRT（gradient boosting regression trees）[13] 和 λ -MART (lambda-multiple additive regression tree) [14]。这两种算法都产生一个附加的决策树集合体模型。所以，下面将回顾一些最先进的决策树遍历算法，这些算法将作为本文算法比较的基线。

a)StructPlus[8]。它是在 STRUCT 算法上的改进。它使用一种数据结构来实现树的遍历。这个数据结构存储了右左孩子节点的指针、特征的下标和阈值。这个遍历过程从一棵决策树的根节点开始。根据遍历到的分支节点上的布尔条件结果，从根节点移动到叶子节点。这个叶子李点的输出值表示了这个树对文档得分的潜在贡献。这种算法的缺点是：StructPlus 频繁地引入控制依赖。也就是说只有在布尔条件测试之后，才能知道下一个遍历的节点。所以下一条执行的指令依赖于布尔条件的结果。这种算法的效率与分支误预测率有很大的关系。

b)PRED[8]。Asadi 等人提出一种重排列方法。它将控制依赖转换成数据依赖。PRED 算法将一棵树转换为一个数据结构类型的数组(Node)。Node[i]表示树上的一个分支或叶子节点。它存储了特征下标（fid）、阈值（theta）和一个用于保存左右孩子节点下标的数组（child）,第一个变量 child[0]，保存当前节点的左孩子节点下标位置。Child[1]保存当前节点的右孩子的下标位置。为了得到下一个将要遍历节点的下标，将利用式（1）的输出值：

$$v[nide[i].fid] \succ node[i].theta \tag{1}$$

式（1）的输出值（0|1）被用于 child 数组的下标。本文使用自循环的方式去连接每一个叶子节点到它本身。一个深度为 d 的树被展开成 d 个操作：

$$d \quad depth\begin{cases} i = node[i].child[(v[node[i].fid] > node[i].theta)] \\ i = node[i].child[(v[node[i].fid] > node[i].theta)] \\ . \\ . \\ i = node[i].child[(v[node[i].fid] > node[i].theta)] \end{cases} \tag{2}$$

这导致了即使遍历过程提前到达叶子节点，遍历过程也不能提前退出，必须完成 d 个操作。除此之外，它还包含很长的展开代码。

c)VPRED[8]。它是在 PRED 算法上进行的改进。它使用一个向量化的方法去计算多篇文档的得分来减少指令分支的误预测率和掩盖内存的高访问延迟。VPRED 算法将并行计算 V 篇文档。

2　相关定义和剪枝决策树模型

本章将展现本文使用的一些决策树相关定义，并介绍这个剪枝决策树模型的细节。

2.1　相关定义

一个查询文档对 (q, d_i) 由一个真值特征向量 x 表示，且 $x \in R^{|F|}$。$F = \{f_0, f_1, \ldots\}$ 是描述文档特征的集合。$x[i]$ 存储 f_i 特征值。一棵树 $T = (N, L)$ 包含一组分支节点 $N = \{n_0, n_1, \ldots\}$ 和一组叶子节点 $L = \{l_0, l_1, \ldots\}$。每一个分支节点 $n \in N$ 都由一个布尔条件，一个特征下标 ϕ，且 $f_\phi \in F$ 和一个阈值 $\gamma \in R$ 组成。这个布尔条件的公式为 $x[\phi] \le \gamma$。每一个叶子节点 $l \in L$ 都存储一个预测值 $l.val \in R$，用于表示这棵树对文档 x 的潜在贡献值。假设文档 x 的部分特征值为 $x[1] \le \gamma_0$，$x[0] > \gamma_1$，$x[3] > \gamma_4$。图 1 显示了树的遍历过程。

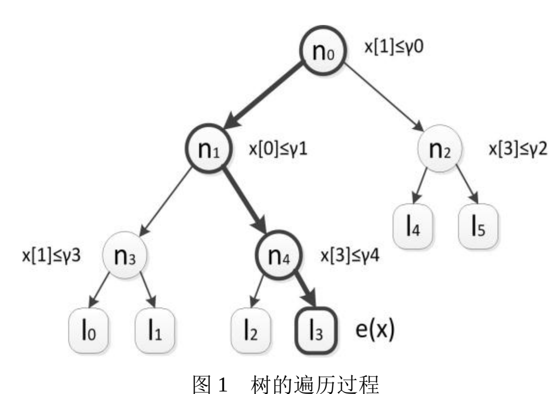


图 1　树的遍历过程

Fig.1 Traversal process of tree

定义 1　假节点和真节点。如果一个分支节点的布尔条件测试为 FALSE，这个节点就是假节点，否则就是真节点。在图 1 中，n_0 是真节点，n_1 和 n_4 是假节点。

定义 2　退出节点。树的遍历过程从根节点开始。如果遍历到的节点是一个假节点（定义 1），则遍历当前节点的右分支，否则遍历左分支。这个遍历过程将递归进行直到到达某个叶子节点。称这个叶子节点为退出节点，使用 $e(x)\in L$ 表示。图 1 中，叶子节点 l_3 是这棵树的退出节点。

定义 3　查询文档对 x 的子得分。这个子得分表示某一棵树 T 对文档 x 的潜在贡献值。这个子得分将会用到定义 2 中的退出节点。计算公式如下：

$$subscore(x)=w\times e(x).val \tag{3}$$

其中：$e(x).val$ 表示树 T 中退出节点的预测值。树 T 权重是 $w\in R$。

定义 4　文档最终得分。这个得分用于排序文档。所有在集合体 $\Gamma=\{T0,T1,...\}$ 中的树都进行这个遍历过程。这个最终得分用到定义 3 中的子得分。文档 x 的最终得分 $s(x)$ 通过对集合体 Γ 中的每一棵树 T 的贡献值进行加权求和得到。

$$\begin{aligned}s(x)&=\sum_{h=0}^{|\Gamma|-1}subscore_h(x)\\&=\sum_{h=0}^{|\Gamma|-1}w_h\times e_h(x).val\end{aligned} \tag{4}$$

2.2　剪枝决策树模型

GBRT 和 λ-MART 算法都产生一个附加决策树集合体。这有一个开源实现了上面两种算法[15]。对于排名文档而言，尽管这些模型可以得到较高的召回率和准确率，但是响应时间非常的长。这是因为排名时间与决策树的个数成正比关系。因此，本文提出了一个剪枝的决策树模型。在不影响原始模型的效果下，减少模型中决策树的个数。

本节主要介绍这个剪枝决策树模型的一些细节。得到这个剪枝决策树模型分为两个步骤：a) 使用 GBRT 或 λ-MART 算法产生一个较好的模型；b) 使用剪枝策略减少原始模型中决策树的个数。

第一步，本文首先使用 GBRT 算法产生决策树模型。GBRT 通过最小化均方根误差来构建决策树模型。GBRT 使用一个训练集 $\{(x_1,y_1),\ldots\ldots,(x_n,y_n)\}$。一个损失函数 $L(y,F(x))=|y-F(x)|$ 和迭代次数 M。这种算法首先使用一个常量进行初始化。

$$F_0(x)=\arg\min_{\gamma}\sum_{i=1}^{n}L(y_i,\gamma) \tag{5}$$

然后迭代 M 次去构建 M 棵树，以提高决策树模型的质量。计算一个伪残差：

$$\begin{aligned}\tilde{y}_i&=-\left[\frac{\partial L(y_i,F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}\\&=\mathrm{sign}(y_i-F_{m-1}(x_i))\end{aligned}$$

$$\text{for } i=1,\ldots,n \tag{6}$$

这个训练集 $\{(x_i,y_i)\}_{i=1}^{n}$ 是用于训练这个学习器。通过求解下面一维优化问题来计算：

$$\begin{aligned}\gamma_m&=\arg\min_{\gamma}\sum_{i=1}^{n}L(y_i,F_{m-1}(x_i)+\gamma h_m(x_i))\\&=\arg\min_{\gamma}\sum_{i=1}^{n}|y_i-F_{m-1}(x_i)-\gamma h_m(x_i)|\\&=\arg\min_{\gamma}\sum_{i=1}^{n}|h(x_i)|\times\left|\frac{y_i-F_{m-1}(x_i)}{h(x_i)}-\gamma\right|\end{aligned} \tag{7}$$

最后更新决策树集合体：

$$F_m(x)=F_{m-1}(x)+\gamma_m h_m(x) \tag{8}$$

第二步，本文使用一种剪枝技术去剪枝刚刚得到的模型。给定一个集合体 Γ，剪枝技术将产生一个更小的集合体 Γ_p，且至少与原来集合体 Γ 有相同的质量，但有更高的效率。剪枝技术分为两步：a) 使用剪枝策略将 Γ 剪枝成一个子集 Γ_p；b) 使用线性搜索算法更新剩下树 Γ_p 的权重值，以改善一个指定的损失函数。

首先使用剪枝策略删除集合体中一些树。这一步使用一个叫质量损失的剪枝策略去减少决策树的个数。

定义 5　质量损失。根据每一棵树对一个指定的损失函数 L 的影响去删除树。对于一个树 T_i，平均质量损失值是通过等式 $s(x)-subscore_i(x)$ 来计算，其中 $subscore_i(x)$ 和

$s(x)$ 在定义 3 和 4 中。影响最小的 n-p 棵树就会从集合体中删除。

然后使用线性搜索算法调整剩下树 Γ_p 的权重值。$L:R^P \to R$ 表示一个指定的损失函数，$\Lambda = \{w_1, w_2, ..., w_p\}$ 表示在剪枝之后树的权重值。使用式（9）对权重值进行优化：

$$\overline{\Lambda} = \underset{\Lambda \in R^P}{\arg\min}\, L(\Lambda) \tag{9}$$

因为大多数损失函数是不可微分，计算 $\overline{\Lambda}$ 是不太可能的，所以使用一种启发式方法去优化剩余树的权重值 Λ。首先找到一个沿着损失函数 L 下降方向 $D \in R^P$。然后计算一个步长 α 以便最小化 $L(\Lambda + \alpha \times D)$ 值。这个过程将一直迭代，直到小于指定的阈值。

3　剪枝和基于分块的方法

本章将展现这个剪枝和分块算法的方法并描述其中细节。研究者们已经证明排名学习模型可以有效的解决排名问题，与此同时，响应时间也很长。为了使这些有效的模型应用到现代搜索引擎中，本文提出使用剪枝和缓存两种技术去减少排名时间的方法。希望通过使用这种方法可以将排名模型应用到现代搜索引擎中。

3.1　剪枝和分块方法

本节主要描述该方法的结构。图 2 描述了方法的大体流程。这种方法的过程可以分成三步：a) 使用 GBRT 或 λ-MART 算法从训练集中训练出一个决策树模型，这一步对应图 2 中标 1 的边；b) 使用剪枝策略去减少排名模型中树的个数并更新剩余树的权重值[16]，这一步对应图 2 中标 2 的边；c) 使用缓存技术去计算所有文档的得分，然后使用这些得分排名文档，这一步对应标 3 的边。

本文提出的方法可以减少排名时间主要有两个原因：第一，计算文档得分的时间与决策树个数成正比，当使用剪枝技术去减少树的个数时，计算文档得分时间就会减少；第二，缓存的访问延迟比内存快几百倍，当算法需要的文档和决策树在缓存中，计算文档得分的时间就会进一步减少。

这个剪枝和基于分块的方法分为两个部分：第一是剪枝过程。对应图 2 左上方。该剪枝过程在不影响原始模型的效果下，减少模型中决策树的个数。所以，本文使用这个剪枝决策树模型作为最终的排名模型。3.2 节中介绍了这个剪枝决策树模型。第二是缓存方案，对应图 2 左下方。下面的章节主要描述基于分块的算法。

3.2　分块技术

本节主要描述基于分块算法实现的细节。本文提出使用分块方案主要是因为缓存访问延迟比内存快几百倍，所以希望能从缓存中获取文档和排名器（决策树）。但是缓存通常非常小，缓存不能包含所有的候选文档和模型中所有的排名器，因此缓存中的内容被频繁的替换，这将导致缓存时间局部性的利用率非常低。为了改善缓存时间局部性的利用率，本文提出基于分块算法。

分块算法的主要思想是将一个大的候选文档集和排名模型分别划分到各自的几个块中，一个文档块和一个排名器块可以同时放入到缓存中。

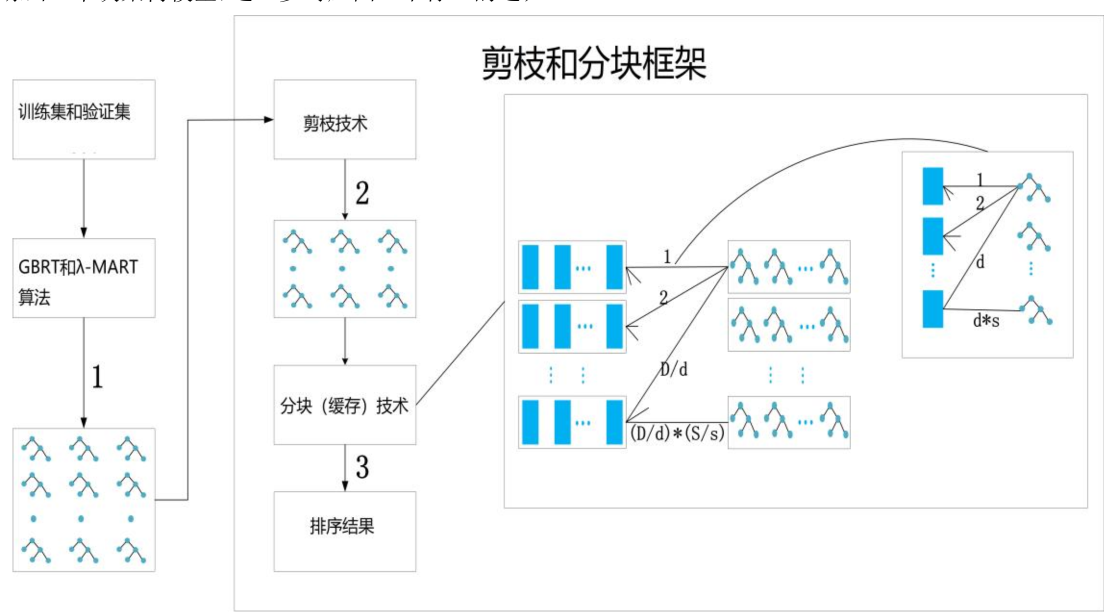


图 2　剪枝和分块方法

Fig.2　Pruning and block-wise approach

基于分块算法实现细节：该算法使用四个嵌套的循环语句。最外层的两个循环用于处理文档块与排名器块之间的交互，而最内层的两个循环用于处理一个文档块中的文档与一个排名器块中的排名器之间的交互。假设候选文档集中包含 D 篇文档，决策树集合体中包含 S 棵排名器。一个文档块中包含 d 篇文档，

一个排名器块中包含 s 棵排名器。一共有 $\frac{D}{d}$ 个文档块和 $\frac{S}{s}$ 个排名器块。为了简化算法的描述，假设 $\frac{S}{s}$ 和 $\frac{D}{d}$ 都是整数。候选文档与排名器的遍历过程显示在图 2 的右半部分，且边上的数字表示了文档块与排名器块之间的访问顺序。

根据上面的分析，本文给出算法 1。

算法 1　分块算法.

Input:

documents：候选文档集;

ensemble　：决策树集合体;

D　　　：候选文档中的文档个数;

d　　　：一个文档块中文档的个数;

S　　　：决策树集合体中决策树的个数;

s　　　：一个决策树块中决策树的个数;

Output:

scores　：文档得分的集合体，一篇文档一个得分

for i = 0 to $\frac{S}{s}$ - 1 do

　for j = 0 to $\frac{D}{d}$ - 1 do

　　for ii = 0 to s - 1 do

　　　for jj = 0 to d-1 do

$$subscore(x_{j\times d+jj}) = w_{i\times s+ii} \times e_{i\times s+ii}(x).val$$ (见定义 3)

$$scores(x_{j\times d+jj}) += subscore(x_{j\times d+jj})$$ (见定义 4)

　　　end for

　　end for

　end for

end for

4　实验

在两个公共可用的数据集上进行了一系列实验。这两个数据集是 Microsoft LETOR （MSLR-10K）（Microsoft Learning to rank dataset. http://research.microsoft.com/en-us/projects/mslr）和 Istella 提供的一个新数据集（Istella-s）（Istella Blog, Istella Learning to Rank dataset. http://blog.istella.it/istella-learning-to-rank-dataset/）。第一个数据集包含五个文件夹，本文的实验仅使用了第一个文件夹（MSN-1）。MSN-1 数据集中每一个查询文档对由 136 个特征表示。Istella-s 数据集包含 3 408 630 个查询文档对。每一个文档使用 220 个特征来表示。每一个查询文档对都有一个相关性判断级别，范围从 0（不相关）～4（非常相关）。所有数据集都被划分成训练集、验证集和测试集三个部分。所占比例分别是 60%-20%-20%。

所有实验运行在一个 Linux 计算机上，它由一个 2 核 Intel 的 I3-6100 CPU，且主频是 3.7 GHz 和一个 8 GB 的内存组成。CPU 缓存级别：L1 数据缓存，且每一个核是 32 KB；L2 的缓存容量是每一个核是 256 KB 和所有核共享的 3 MB 的 L3 缓存。

在实验中使用 NDCG@10 对训练的模型进行效果评估[17]。

$$\mathrm{NDCG@10} = \mathrm{N}^{-1}\sum_{j=1}^{10} g(r_j)d(j) \tag{10}$$

其中：N^{-1} 是一个归一化的因子，表示一个最理想的排名结果的 DCG 得分；r_j 表示第 j 个文档的相关性级别；$g(r_j)$ 是一个幂函数：

$$g(r_j) = 2^{r_j} - 1 \tag{11}$$

d(j)通过式（12）计算：

$$d(j) = \frac{1}{\log_2(1+j)} \tag{12}$$

在本文的实验中，将剪枝和分块算法与下面的算法在效率和效果上进行比较：

a）标准遍历算法。一篇文档直接遍历所有的树。

b）不使用缓存技术。

c）StructPlus 和 VPRED[8]。

笔者进行了下面的三组实验来分析剪枝和分块算法的效率和效果。

4.1　效果分析

本文框架首先使用剪枝技术去删除集合体中一些树，然后更新剩余树的权重。所以，首先评估剪枝之后模型的效果。这组实验中进行两个子实验，分别使用 GBRT 和 λ-MART 算法从两个公开的数据集（MSN-1 和 Istella-s）训练出一个排名模型。为了更好地分析不同参数值对 NDCG@10 的影响，在每一个子实验中使用不同的剪枝率和叶子节点个数。设置剪枝率 P 为 0，0.1，0.2，……，0.9，且叶子节点的个数为 8,16,32,64。当 P=0 时，这个 NDCG@10 的值表示原始模型的效果得分，即剪枝之前模型的效果评估值。两个子实验的结果显示在图 3 中。

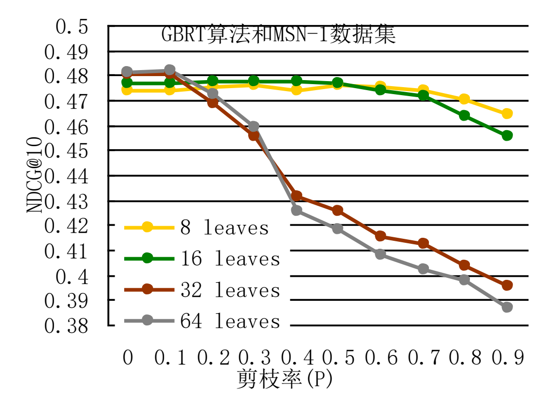


图 3(a) 使用 GBRT 算法和 MSN-1 数据集在不同叶子节点和剪枝率下的 NDCG@10 的评估值

Fig.3(a)　NDCG@10 in vary leaf and pruning rate with GBRT algorithm and MSN-1 dataset

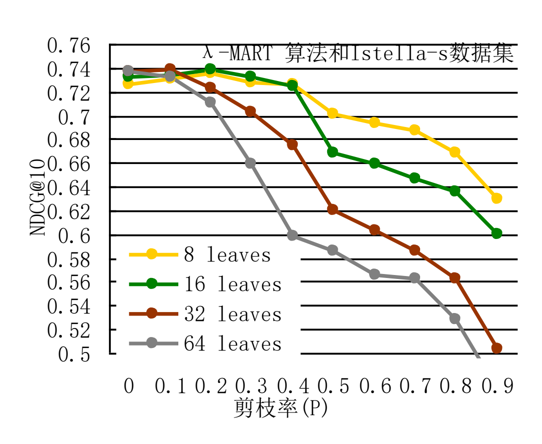


图 3(b)　使用 λ-MART 算法和 Istella-s 数据集在不同叶子节点和剪枝率下的 NDCG@10 的评估值

Fig.3(b)　NDCG@10 in vary leaf and pruning rate with λ-MART algorithm and Istella-s dataset

从上面两个子实验中可以看出，在不同的剪枝率下，具有较少叶子节点（8,16）的模型比具有较多叶子节点（32,64）的模型有更好的效果评估，且较少叶子节点的模型的效果比较多叶子节点的模型的效果下降得更慢。本选择这个最小的（即最有效率的）剪枝模型，但仍然提供比原始模型在验证集中相等或更大的效果评估。最终选择的剪枝模型显示在表 1 中。

表 1 显示在不同叶子节点和数据集下最终选择的剪枝决策树模型。从剪枝率的方面来看，高亮和加粗的部分表示最好的剪枝模型。从表 1 可以看出，通过使用剪枝技术，可以显著地减少集合体中 30%到 70%的决策树，且不影响原始模型的效果；还可以观察到具有较少的叶子的模型中有更高的剪枝率。因此，本文选择高亮和加粗的剪枝决策树模型作为下面实验的模型。

表 1　不同设置下的效果评估

Table 1　Effectiveness under different setting

算法和数据集	叶子节点个数											
	8			16			32			64		
	树个数	剪枝率	NDCG@10	树个数	剪枝率	NDCG@10	树个数	剪枝率	NDCG@10	树个数	剪枝率	NDCG@10
GBRT 和 MSN-1 未剪枝	777	---	0.4734	903	----	0.4764	553	---	0.4800	470	---	0.4811
GBRT 和 MSN-1 剪枝	**233**	**0.7**	**0.4735**	**451**	**0.5**	**0.4767**	498	0.1	0.4805	423	0.1	0.4816
λ-MART 和 Istella-s 未剪枝	999	---	0.7263	672	---	0.7317	695	---	0.7371	604	---	0.7373
λ-MART 和 Istella-s 剪枝	**600**	**0.4**	**0.7264**	**470**	**0.3**	**0.7321**	625	0.1	0.7389	604	0	0.7373

4.2　缓存行为分析

本文算法使用了缓存技术去减少排名文档的时间。这种基于分块的算法将一个候选文档集和决策树集合体分别划分到各自几个块中。一个包含 d 个文档的文档块和一个包含 s 个排名器的排名器块被放入到缓存中。本文将分析不同文档个数 d 和不同排名器个数 s 对 L3 缓存的失误率和排名时间的影响。Linux 的 Perf 工具分析 L3 缓存的未命中率。

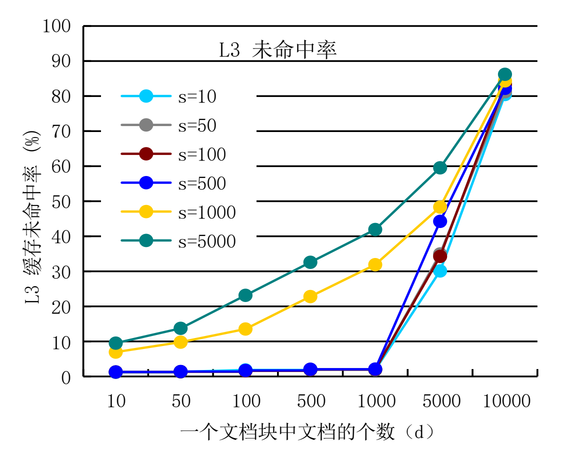


图 4　(a)　显示 L3 在不同的 d 和 s 下的未命中率

Fig.4(a)　L3 miss ratio when varying d and s for Block-wise algorithm

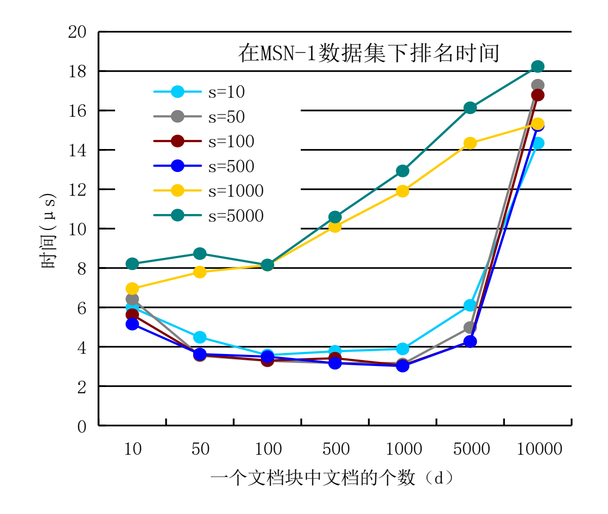


图 4　(b)　在不同 d 和 s 下,MSN-1 数据集中计算一篇文档得分的平均时间/μ

Fig.4(b)　ranking time per document in μs when varying d and s under MSN-1 datasets

图 4（a）显示了分块算法在不同的 d 和 s 下 L3 缓存的未命中率。这个趋势与图 4（b）中排名时间曲线有很大的关系。图 4（b）显示在不同的 d 和 s 下，计算一篇文档得分需要的时间。从图 4（a）和（b）中可以看出，当 d=1000 和 s=500 时到达了这个最优点，即这些文档和决策树正好放入到缓存中，并

且循环次数也降到最低；当 s=500 时，L3 的未命中率从 2.11%（d=1000）变化到 82.22%（d=10000），因此，对应的排名时间从 3.0191μs 变化到 15.2335μs。在下面的实验中将设置 d=1000 且 s=500。

4.3　时间复杂度的分析

在本节中将分析各个排名模型的时间复杂度，以形式化地观察各个排名模型的排名效率。

本算法的时间复杂度主要是 3.2 节中的算法 1，该算法利用缓存技术将文档集和决策树进行分块。利用缓存速度快的优点，提高排名模型的效率。

算法 1 主要有四层循环。为了简化描述，这里假设 s 个决策树和 d 个文档可以同时放入到缓存中。在最内层循环中，将 d 篇文档从内存加载到缓存的时间为 $O(d)$。在第三层循环中，将 s 个决策树加载到缓存的时间为 $O(s)$。缓存可以同时容下 d 篇文档和 s 个决策树。所以这个两层的时间复杂度为 $O(s+d)$。在第二层循环中，s 个决策树仍然在缓存中，但要从内存中将 $\frac{D}{d}$ 块文档加载到缓存中，因此时间复杂度为 $O(s)+\frac{D}{d}O(d)=O(s+D)$。在最外层循环中，在加载 $\frac{S}{s}$ 块决策树到缓存中，因此算法 1 的总时间复杂度为 $\frac{S}{s}O(s+D)=O(S+\frac{SD}{s})$。

在不使用缓存技术算法、标准遍历算法和 StructPlus 算法中，它们都是一篇文档和一个决策树进行遍历，因此这三种算法的时间复杂度为 $O(SD)$。本文算法相比这三种算法快了 s 倍。而 Vpred 算法采用了并行化的技术，同时计算 v 篇文档的得分，使用并行化技术来掩盖低内存的访问延迟。在文献[8]中表明当 v=16 时，该算法表现的效果最好，因此算法的时间复杂度为 $\frac{O(SD)}{16}$。从 4.2 节中得到 $s>>16$。从时间复杂度上看，算法 1 比其他算法的时间复杂度都要低。

在 4.4 节中将进行四组实验来验证各个排名模型在不同数据集上的排名时间。

4.4　排名时间的分析

本文的目的是减少排名时间以将有效果的排名模型应用到搜索引擎中。所以本节中将剪枝和分块算法与其他排名算法在效率方面进行比较。这组实验中进行了四个子实验。

表 2 和 3 中分别显示了 GBRT 算法和λ-MART 算法在 MSN-1 和 Istella-s 数据集下不同的遍历算法计算一篇文档得分的时间（μs）。表 2 和 3 同样在括号中显示剪枝和分块算法比其他算法的增速比。例如，在计算一篇文档的得分时间上，本文算法比 VPRED（v=16）算法快了 1.17～2.24 倍。在 MSN-1 数据集和叶子节点个数为 8 下（表 2），本文算法和 VPRED 算法计算一篇文档得分时间分别为 3.0191μs 和 6.79μs。从表 2、3 中可以看出，在不同的设置下，剪枝和分块算法在效率方面比其他算法表现得要好，显示了本文算法比其他算法在排名时间上快了 1.17～10.35 倍。

表 2　MSN-1 数据集且 GBRT 的决策树模型下,剪枝和分块算法、不使用缓存、VPRED、StructPlus 和标准遍历算法排名一篇文档的时间/μs

Table 2　Per-document scoring time (μs) of pruning and block-wise, No use cache, VPRED, StructPlus, and reference on MSN-1 dataset under GBRT decision tree model

遍历算法	叶子节点个数	
	8	16
剪枝和分块算法 (d=1000 and s=500)	**3.0191(剪枝率 0.7)**	**8.4304(剪枝率 0.5)**
不使用缓存	7.3224(2.43x)	20.6485(2.45x)
标准遍历	31.2462(10.35x)	80.6496(9.57x)
StructPlus	22.68(7.51x)	55.01(6.53x)
VPRED(v=16)	6.79(2.24x)	15.10(1.79x)

表 3　在 Istella-s 数据集且λ-MART 模型下,剪枝和分块算法、不使用缓存、VPRED、StructPlus 和标准遍历算法排名一篇文档的时间

Table 3　Per-document scoring time (μs) of pruning and block-wise, No use cache, VPRED, StructPlus, and reference on Istella-s dataset under λ-MART

遍历算法	叶子节点个数	
	8	16
剪枝和分块算法 (d=1000 and s=500)	**7.1993(剪枝率 0.4)**	**9.2954(剪枝率 0.3)**
不使用缓存	22.5395(3.13x)	38.8463(4.17x)
标准遍历	37.8187(5.25x)	57.6416(6.20x)
StructPlus	28.36(3.94x)	35.49(3.82x)
VPRED(v=16)	8.71(1.21x)	10.86(1.17x)

5　结束语

现代搜索引擎对响应用户查询有严格的时间限制。本文提出了一种方法去减少排名候选文档的时间，以将高质量的排名模型用于现代搜索引擎中。这种方法整合了剪枝决策树模型和分块算法。在相同的时间限制下，该方法可以排名更多文档，从而获得更好的召回率和准确率。实验结果表明了该方法可以在不影响模型的质量下，有效地减少每个查询所需的排名时间。对于将来的工作，笔者打算应用这个框架到其他方面，如分类等方向。

参考文献：

[1] Liu Tieyan. Learning to rank for information retrieval [J]. Foundations and Trends in Information Retrieval, 2009, 3 (3): 225-331.

[2] 王扬, 黄亚楼, 刘杰, 等. 基于 PRank 算法的主动排序学习算法 [J]. 计算机工程, 2008, 34 (21): 38-39. (Wang Yang, Huang Yalou, Liu jie, *et al*. Algorithm of active learning to rank based on PRank algorithm [J]. Computer Engineering, 2008, 34 (21): 38-40.)



[3] Ganjisaffar Y, Caruana R, Lopes C V. Bagging gradient-boosted trees for high precision, low variance ranking models [C]// Proc of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM Press, 2011: 85-94.

[4] Viola P, Jones M. Robust real-time face detection [J]. International Conference on Computer Vision, 2001, 57 (2): 137-154.

[5] Cambazoglu B B, Zaragoza H, Chapelle O, *et al.* Early exit optimizations for additive machine learned ranking systems [C]// Proc of the third ACM International Conference on Web Search and Data Mining. New York: ACM Press, 2010: 411-420.

[6] Robertson S, Zaragoza H. The probabilistic relevance framework: BM25 and beyond [J]. Foundations and Trends in Information Retrieval, 2009, 3 (4): 333-389.

[7] Lucchese C, Nardini F M, Orlando S, *et al.* QuickScorer: a fast algorithm to rank documents with additive ensembles of regression trees [C]// Proc of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM Press, 2015: 73-82.

[8] Asadi N, Lin J J, De Vries A P. Runtime optimizations for tree-based machine learning models [J]. IEEE Trans on Knowledge and Data Engineering, 2014, 26 (9): 2281-2292.

[9] Lucchese C, Nardini F M, Orlando S, *et al.* Speeding up document ranking with rank-based features [C]// Proc of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM Press, 2015: 895-898.

[10] Lucchese C, Nardini F M, Orlando S, *et al.* Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles [C]// Proc of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM Press, 2016: 833-836.

[11] 周栋，赵文玉，伍璇，等. 个性化跨语言信息检索中结果重排序研究 [J]. 计算机工程与科学，2017, 39 (10): 1922-1929. (Zhou Dong, Zhao Wenyu, Wu Xuan, *et al.* Result re-ranking in personalized cross-language information retrieval [J]. Computer Engineering and Science, 2017, 39 (10): 1922-1929.)

[12] Tang Xun, Jin Xin, Yang Tao. Cache-conscious runtime optimization for ranking ensembles [C]// Proc of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM Press, 2014: 1123-1126.

[13] Friedman J H. Greedy function approximation: a gradient boosting machine [J]. Annals of Statistics, 2001, 29 (5): 1189-1232.

[14] Wu Qiang, Burges C J, Svore K M, *et al.* Adapting boosting for information retrieval measures [J]. Information Retrieval, 2010, 13 (3): 254-270.

[15] Capannini G, Lucchese C, Nardini F M, *et al.* Quality versus efficiency in document scoring with learning-to-rank models [J]. Information Processing and Management, 2016, 52 (6): 1161-1177.

[16] Lucchese C, Nardini F M, Orlando S, *et al.* Post-learning optimization of tree wnsembles for efficient ranking [C]// Proc of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM Press, 2016: 949-952.

[17] Jarvelin K, Kekalainen J. Cumulated gain-based evaluation of IR techniques [J]. ACM Trans on Information Systems, 2002, 20 (4): 422-446.